Test Case Generation for Simulink Models using Model Fuzzing and State Solving

Zhuo Su†, Zehong Yu†, Dongyan Wang‡, Wanli Chang§, Bin Gu¶, Yu Jiang🖂 †
† KLISS, BNRist, School of Software, Tsinghua University, Beijing, China
‡ Information Technology Center, Renmin University of China, Beijing, China
§ College of Computer Science and Electronic Engineering, Hunan University, Changsha, China
¶ Beijing Institute of Control Engineering, Beijing, China

ABSTRACT

Simulink plays an important role in the industry for modeling and synthesis of embedded systems. Ensuring system stability requires using numerous test cases to validate the functionality and safety of the models. However, as requirements increase, the complexity of the models poses new challenges to traditional testing methods. Traditional methods such as constraint solving and random search run into significant obstacles when navigating the complex branching logic and states within models.

In this paper, we introduce `HybridTCG`, a test case generation method by collaborating model fuzzing and state solving for Simulink models. First, `HybridTCG` starts a code-based fuzzer to generate high-coverage test cases rapidly. Then, it refines the test cases generated by the fuzzer, preserving only those that can achieve new model coverage. These selected test cases are input into the state-solving engine to derive corresponding states and resolve the constraints of subsequent branches. Ultimately, the test cases produced by the solving engine will be fed back into the fuzzer as high-quality seeds to enhance the fuzzing process. We have implemented `HybridTCG` and conducted a comprehensive evaluation using various benchmark Simulink models. Compared to the built-in Simulink Design Verifier and state-of-the-art academic work SimCoTest and STCG, `HybridTCG` achieves an average improvement of 54%, 108% and 24% on Decision Coverage, 50%, 62% and 6% on Condition Coverage, 291%, 282% and 45% on Modified Condition Decision Coverage, respectively. Moreover, `HybridTCG` is also much more efficient in testing than other tools.

KEYWORDS

Test case generation, Simulink, Constraint solving, Model fuzzing

1 INTRODUCTION

Simulink is increasingly utilized in embedded design due to its ability to provide efficient modeling, simulation, and high-quality code generation [22, 32, 33, 35, 36, 39]. To ensure that the model meets the functional design requirements and avoids system exception, developers usually need to provide a large number of test cases for testing the model[12]. However, writing test cases manually is a very tedious task. Especially for those deep logics in the model, it is difficult for testers to construct elaborate test cases to trigger them. In recent years, a number of automated test case generation efforts have emerged to effectively reduce the cost of manual testing, while making it relatively easy to explore deep logic [6, 10, 15, 16, 25, 27, 31, 37].

Yu Jiang is the corresponding author.

Broadly speaking, existing test case generation efforts can be categorized into two types. The first is the solving-based method, of which the Simulink Design Verifier (SLDV) [32], a component of the Simulink toolkit, is a good example. In this method, Simulink models are converted into specific formal representations, and then a formal solver is applied to solve for model input values that satisfy the requirements of model coverage. With this method, model input data can be accurately solved to meet specific requirements. The second is the search-based method, typically using tools such as SimCoTest [25]. In this method, input data for the model is randomly generated and executed to obtain coverage feedback. Such feedback information is then used to further refine and optimize the test case generation process. This approach can be relatively quick to achieve high model coverage.

It is undeniable that previous efforts have brought good benefits to automatic test case generation, but as requirements become increasingly complex and model scales expand rapidly, these efforts also face challenges. Specifically, these challenges mainly stem from the growth of model states and the increasing complexity of control logic. Traditional constraint-solving methods suffer from a severe state explosion problem, which makes it difficult to obtain feasible solutions within limited time and computational resources. As for search-based methods, the probability of generating a specific model state required by a control condition is extremely low.

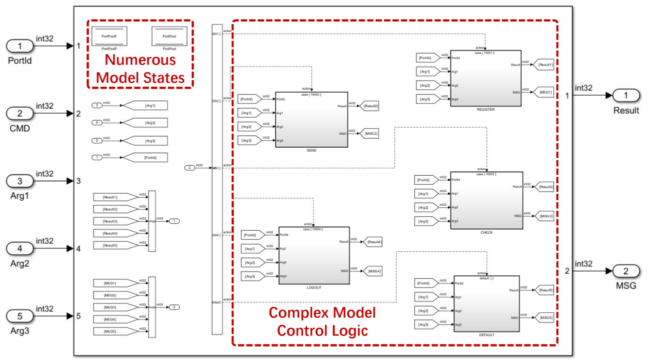


Figure 1: An example model with numerous states and complex control logic. This is a LAN Switch model. It mainly includes the registration, checking, data sending and logout functions of network devices.

To better illustrate the aforementioned challenges, we present the LAN switch model in Figure 1, which allows multiple devices on a local area network to communicate with each other. The main functions of the model include port registration, port checking, data sending, and port logout. Since the switch needs to connect multiple device ports, the model requires a port pool and a port state pool to keep tracking them. To achieve data exchange, especially in the data-sending function, the model must find the appropriate destination ports and perform specific functions according to their states. These two basic requirements dictate that the model is complex in both state and control logic. For the constraint-solving methods, it is difficult to generate a simple test case of two devices communicating. This is because several preparations such as "registering ports" and "establishing connections" must be completed before the two devices can send data to each other. Constraint-solving methods are exponentially more complex in terms of state space and logic complexity to generate this series of operations. For search-based methods, although most of these methods use coverage feedback to optimize exploration, it is still unrealistic to explore a given set of logic in a finite amount of time by random mutation.

To address the problems faced by the aforementioned approaches or methods, we propose `HybridTCG`, a test case generation technique that collaborates the model fuzzing method and the state-solving method. The fuzzer quickly generates test cases with high coverage and reduces a large number of solving tasks for the state-solving engine. On the other hand, the state-solving engine also provides more high-quality seeds for the fuzzer. In detail, `HybridTCG` compiles the model fuzz driver, instrumented model code and model input mutator into a coverage-guided fuzzer. Then, the fuzzer is run continuously. At regular intervals, `HybridTCG` refines the test cases generated by the fuzzer by whether they can trigger new coverage or not. The refined test cases are provided to the state-solving engine for execution to obtain corresponding model states. After that, the state-solving engine performs one-step solving of the uncovered branches on the model state to explore deeper model logic. Finally, the solved test cases are provided to the fuzzer for better mutation.

We implemented and evaluated `HybridTCG` on public benchmark Simulink models [37]. Compared to the built-in Simulink Design Verifier and the academic work SimCoTest and STCG, `HybridTCG` achieves an average improvement of 54%, 108% and 24% on Decision Coverage, 50%, 62% and 6% on Condition Coverage, 291%, 282% and 45% on Modified Condition Decision Coverage, respectively. Moreover, `HybridTCG` is also much more efficient than other tools.

The main difference between `HybridTCG` and previous hybrid software fuzzers. The model execution is stateful, i.e. the model is executed one step at a model iteration, and each step results in a new model state that reflects the values of global variables of the model. The main difference is that hybrid fuzzing techniques in the software domain do not care about the execution state of the program, such as Driller[34], SAFL[38] and MEUZZ[8], where constraint solving only uses the execution paths of the test cases generated by the fuzzer or solver to collect constraints. `HybridTCG`, on the other hand, solves the uncovered branches more deeply based on different model states obtained from the fuzzer.

2 BACKGROUND AND RELATED WORK

2.1 Model-driven design.

Model-driven design is an important methodology in embedded software development. It primarily revolves around four core elements: behavior modeling, simulation, testing, and code generation [4, 11, 19, 20, 30]. The process begins with behavior modeling, where a detailed model is created using either text or graphics to precisely match user requirements. Simulation is critical in the design phase, allowing debugging and verification of the model's functionality. Testing is also an essential step, providing a systematic approach to thoroughly validating the model and ensuring its reliability. The final step, code generation, transforms the model into executable code for use in embedded devices, turning theoretical designs into practical solutions. Simulink by MathWorks stands out as the most popular tool. It offers powerful simulation and code generation features. Additionally, Simulink provides a comprehensive component library, serving various industries efficiently, making the system design and development process highly efficient.

2.2 Model execution and test case

To better understand test case generation, a clear understanding of the model execution process and the test case format is necessary.

Model execution process. In the embedded domain, the control logic of a device is usually executed cyclically according to a fixed clock cycle. A single execution of the control logic is often referred to as a "step", and the execution of the Simulink model follows a similar paradigm. Each step reads data from inports and then performs a series of calculations and passes the data to outports. This is equivalent to an embedded device reading and writing data from pins. In control systems, some system states are usually set during the execution of step. These states will have an impact on the subsequent step execution process or results. For example, a traffic light control system records the current light state for the countdown and color switching. Accordingly, the Simulink model is executed in a similar way as described above.

Test case format. When we do model testing, especially coverage-guided testing, getting test data from files is a necessary step. Depending on the execution logic of the model, a test case may contain many "rows". Each row represents one-step input, i.e., the data needed to execute all the inports of the top-level model once. In each row, the data corresponding to the model inports are arranged in order. Below is an example of a test case, corresponding to the model in Figure 1. It contains three steps, register port 1, check port 1, and register port 2. {⟨PortId(i32), CMD(i32), Arg1(i32), Arg2(i32), Arg3(i32)⟩, ⟨0x01, 0x2711, 0x01, 0x00, 0x00⟩, ⟨0x01, 0x2713, 0x01, 0x10, 0x00⟩, ⟨0x02, 0x2711, 0x01, 0x00, 0x00⟩}.

2.3 Model test case generation.

Testing is an indispensable step in the model-driven software development process, for ensuring the target is functionally correct and operationally stable[2, 5, 14, 29]. The adequacy of testing is usually measured by model coverage metrics, including Decision Coverage, Condition Coverage, MCDC coverage, and so on [17, 23, 26, 28]. The more complex the model, the more difficult it is to construct high-coverage test cases. To reduce the cost of manually constructing test cases, testers usually start with automated test case generation tools. They can automatically explore the deeper logic of the model, effectively avoiding complex manual analysis work. Existing tools can be divided into two main categories according to how they work: solving-based test case generation and random search-based test case generation.

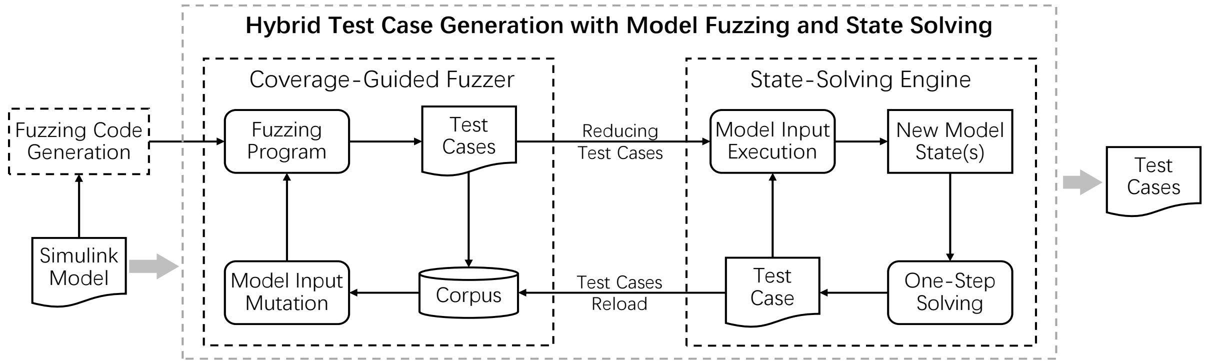


Figure 2: Overview of HybridTCG. Two main parts are executed cyclically to obtain test cases. The coverage-guided fuzzer part focuses on quickly obtaining test cases using model-specific fuzzing techniques. The state-solving engine is used to accurately solve for coverage metrics based on model-specific states. The coverage-guided fuzzer provides more fundamental state space for the state-solving engine. The state-solving engine provides more higher-quality input corpus to the coverage-guided fuzzer. These two parts collaborate to quickly generate high-coverage test cases.

Solving-based test case generation. It usually uses formal techniques to obtain input cases that satisfy the model coverage requirements. Simulink Design Verifier (SLDV), serves as an integral component of Simulink, providing automated test case generation [15]. It extracts the model content related to the coverage metrics by model slicing. It then transforms the sliced model into a formal DSL and obtains test cases from the constraint solver. He et al. demonstrate an approach based on model checking that carefully navigates the architecture of the model [16]. It can identify the subset of nodes that maximize the observation of mutants, and then use this information to generate a concise set of test cases to achieve high coverage. Another innovative strategy is presented in AutoMOTGen by Mohalik et al. which leverages the formal SAL language [18] to articulate Simulink model parameters [27]. This method integrates coverage metrics directly into the formal representation of the model and leverages existing model-checking mechanisms to simplify the test case generation process. A recent work STCG maintains a state tree to hold various states of the model and solves branches based on the state nodes [37]. It uses the input data derived from the solver to form random sequences to explore more of the state space.

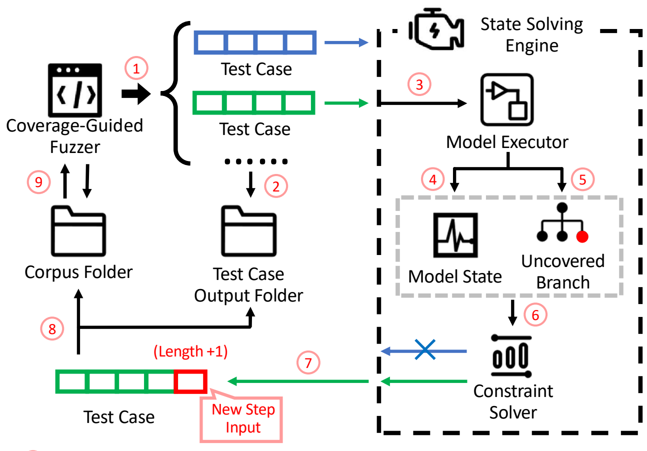
These constraint-solving strategies based on formal methods generally suffer from a lack of efficiency. For more complex models, they may also face the problem of state space explosion, making them difficult to solve. Instead, HybridTCG decomposes the model execution into multiple iterative steps for solving, and introduces fuzzing method to speed up test case generation.

Search-based test case generation. These methods often rely on dynamic simulation of the model using randomized data, and the information obtained during simulation, including execution state and model coverage, is used as valuable feedback to optimize test case generation, as demonstrated by a large number of researchers [3, 10, 25, 31]. Reactis is a prominent example of this method [10]. It uses Monte Carlo random search to generate test cases. It combines this technique with guided simulation. This allows the output values to be examined, effectively assisting in the identification and selection of test cases that have the potential to reveal hidden issues. In a slightly different way, REDIRECT focuses on analyzing the feedback generated by simulated test cases [31]. It utilizes a specific set of heuristics designed for the complex nonlinear blocks. SimCoTest can generate test cases for discrete-time and continuous-time Simulink models. It fully tests the model by maximizing the diversity of output signals.

However, search-based methods still struggle to trigger conditions that depend on complex states and have a low probability of reaching the desired model state. And model-based simulation is also limited by the efficiency of the simulation engine itself. Unlike these approaches, HybridTCG uses a code-based fuzzing approach to maximize the speed of random search. It also leverages constraint-solving methods to continuously provide the fuzzer with high-quality seeds.

3 HYBRIDTCG DESIGN

Figure 2 shows an overview of HybridTCG, which takes Simulink models as input and generates test cases as output. HybridTCG is executed by two main parts that work together, the coverage-guided fuzzer and the state-solving engine. **The coverage-guided fuzzer** is mainly used to quickly generate initial test cases, while it receives the test cases output from the state-solving engine as high quality seeds for further mutation. Specifically, a Simulink model is first transformed into instrumented fuzzing code, which is then compiled together with the fuzz driver and model input mutator into a coverage-guided fuzzer. We then run the fuzzer continuously and extract the test cases it generates at regular intervals to the state-solving engine. As soon as the state-solving engine generates a new test case, it is added to the fuzzer corpus for further mutation. **The state-solving engine** is mainly used to solve one-step model iteration based on the fuzzer's test cases to explore those branches with stringent conditions. Specifically, test cases from the fuzzer are first refined in terms of whether they can trigger new coverage. They are then fed into the model for dynamic execution, with each test case generating a specific internal state of the model. Then, for the uncovered branches, we use the constraint solver to perform a one-step solving on these specific states. Once a new solution is obtained, the solved input of this model iteration is added to the back of the test case for its corresponding state to obtain a complete test case. In order to gain a better understanding of how HybridTCG works, Figure 3 illustrates a workflow.



1. Refine the test cases by whether they trigger new coverage.
2. Output the test cases that trigger new coverage.
3. Bring a test case into the model for execution.
4. Get the model state after the last input step is executed.
5. Random select an uncovered model branch.
6. Solve for the selected branch on the specific model state.
7. Splice the solved step input with the original test case.
8. Export new test cases to the fuzzing corpus.
9. Read in high quality test cases from the corpus.

Figure 3: Workflow of HybridTCG. The collaborative process of the coverage-guided fuzzer and state-solving engine.

3.1 Fuzzing Code Generation

Compared to regular model code generation, the code for model fuzzing requires more content, mainly including fuzz driver generation and model coverage instrumentation.

Fuzz Driver Generation. It is the entry point for model execution and is used to read binary test case files and decompose multiple-step inputs. The main distinction compared to the traditional fuzz driver for software testing lies in that our fuzz driver contains a loop to simulate the execution of the model. Figure 4 illustrates a fuzz driver example that corresponds to the LANSwitch model in Figure 1. Where the first line is the callback function header of the fuzzer, which receives the binary test data. The second line performs the initialization function of the model. Lines 3-18 determine the number of loops to execute the model logic based on the length of data required for one-step execution. Lines 5-9 are variables for model input data. The variables in lines 10-11 are used to save the output of the model. Lines 12-16 are used to get the corresponding input data from the binary data. Finally, the input data is fed in line 17 to execute one step.

Model Coverage Instrumentation. Model coverage is more demanding than traditional code coverage, as testers typically focus on three coverage metrics, Decision Coverage, Condition Coverage and MCDC (Modified Condition/Decision Coverage) [24]. Decisions refer to judgments in the model that trigger different execution logic. For example, switches and state machine transitions. The analogy to code is the different branches. Conditions refer to calculations in the model that trigger different boolean outcomes. For example, Boolean operations, comparison operations, etc. MCDC

```
1  void TestOneStepInput(unsigned char *data, int size){
2    LanSwitch_init();
3    int dataLen = 20;
4    for(int offset = 0; offset + dataLen <= size;){
5      int32 PortId = {};
6      int32 CMD = {};
7      int32 Arg1 = {};
8      int32 Arg2 = {};
9      int32 Arg3 = {};
10     int32 Result;
11     int32 MSG;
12     memcpy(&PortId, data + offset, 4); offset += 4;
13     memcpy(&CMD,    data + offset, 4); offset += 4;
14     memcpy(&Arg1,   data + offset, 4); offset += 4;
15     memcpy(&Arg2,   data + offset, 4); offset += 4;
16     memcpy(&Arg3,   data + offset, 4); offset += 4;
17     LanSwitch_step(PortId, CMD, Arg1, Arg2, Arg3,
                      &Result, &MSG);
18   }
19 }
```

Figure 4: Example fuzz driver for LANSwitch model in Figure 1. Five variables in lines 5-9 are model inport variables. They are passed to the LanSwitch_step function for model one-step execution.

considers whether the reversal of each sub-condition in a decision can independently affect the overall decision outcome.

To better generate test cases that trigger the above metrics in coverage-guided fuzzing, we need to insert the relevant metrics statistics code based on the logic code generated from the model. These instrumented codes can enhance fuzzer's exploration of model coverage metrics. Figure 5 exemplifies the instrumentation details for the expression "$if(A\&\&B||C)$", containing two Decision Coverage metrics, six Condition Coverage metrics and six MCDC metrics. The code will be instrumented in front of the "$if(A\&\&B||C)$" expression. In this case, the first three lines calculate the Condition Coverage metrics. Lines 4-6 statistic the Decision Coverage metrics. Finally, lines 7-15 count the MCDC metrics.

```
1  if(A){StatisticCond(1);}else{StatisticCond(2);}
2  if(B){StatisticCond(3);}else{StatisticCond(4);}
3  if(C){StatisticCond(5);}else{StatisticCond(6);}
4  bool exp_1 = A && B;
5  bool exp_2 = exp_1 || C;
6  if(exp_2){StatisticDeci(1);}else{StatisticDeci(2);}
7  bool masked_A = !B || C;
8  bool masked_B = !A || C;
9  bool masked_C = exp_2;
10 if(!masked_A)
11   if(A){StatisticMCDC(1);}else{StatisticMCDC(2);}
12 if(!masked_B)
13   if(B){StatisticMCDC(3);}else{StatisticMCDC(4);}
14 if(!masked_C)
15   if(C){StatisticMCDC(5);}else{StatisticMCDC(6);}
```

Figure 5: Example instrumented code for enhancing coverage. It will be inserted in front of the "$if(A\&\&B||C)$" expression.

3.2 Coverage-Guided Fuzzing

To achieve high coverage test case generation quickly, we introduce the coverage-guided fuzzing technique. Its main operation is akin to traditional software fuzzing. First, the fuzzer reads the initial test cases (initial seeds) from the corpus and generates random data as an initial seed if it is not available. The initial seeds are then passed one by one to the fuzz driver, corresponding to the "TestOneStepInput" function in Figure 4. The fuzzer dynamically executes the model code to obtain coverage information for each test case. Then, the fuzzer prefers to save those test cases that trigger more or new coverage into the corpus. These test cases will be used to mutate to trigger more uncovered branches or paths.

We redesign the mutation strategies to be more adaptable to model-based test case generation. This is because the traditional mutation strategy in terms of binary bytes can seriously affect the alignment of the model's input data sequences. The specific variation strategies of HybridTCG are shown in Table 1.

Table 1: The strategies of model input mutation

Strategy	Description
Change Binary Integer	Modify a binary integer value.
Change Binary Float	Modify a binary float value.
Erase Rows	Remove a range of rows.
Insert Row	Insert a new row with random values.
Insert Repeated Rows	Insert a sequence of repeated rows.
Shuffle Rows	Shuffle the order of rows.
Copy Rows	Copy rows into another position.
Rows Cross Over	Combine rows from two streams.

The "Row" in Table 1 denotes the set of inport data required for one step of the model execution. In particular, for the "Change Binary Integer" and "Change Binary Float" strategies, HybridTCG does not change the data completely at random. Instead, HybridTCG makes finer modifications based on their values. For instance, it can shift an integer, add a constant, or invert a bit. For float numbers, the sign, exponent, and significand bits are mutated more finely according to the IEEE 754 standard, including sign bit inversion, exponent bit modification, etc. The Insert Row and Insert Repeated Rows strategies use a simple dictionary to generate random data.

3.3 State Solving Engine

The state-solving engine is primarily utilized to explore deeper into the state space traversed by the coverage-guided fuzzer. Instead of generating test cases with multiple steps, it adds one-step data that triggers a new coverage to the back of the test cases provided by the fuzzer. Choosing single-step solving for the solver is because compared to multi-step solving, single-step solving is faster and easier to obtain the solution. The state-solving engine consists of two main modules: model input execution and one-step solving. The model input execution module is employed to execute the refined test cases from the fuzzer, obtain the model state after each test case execution, and tally the uncovered metrics. The one-step solving module is used to solve for those uncovered metrics on different model states. The metrics used to be solved by the solver are consistent with the previous fuzzer part and also include Decision Coverage, Condition Coverage, and MCDC. These metrics are encoded as SAT statements to be solved along with the constraints of the model. Once new coverage is explored, the new step input is merged behind the corresponding test case and the new test case will be passed to the fuzzer corpus.

Algorithm 1 shows the exact process of the state-solving engine. Two of the variables in lines 1-2 are used to collect information about the model coverage and the state corresponding to each test case. The refined test cases are then executed separately, in lines 3-6. In lines 7-19, the metrics not yet covered are traversed and these metrics are solved state by state in lines 12-14. Once an input step (*res*) has been solved for a state, we append it to the end of the corresponding test case as a new test case (Length + 1) for output.

To speed up the efficiency of the state-solving engine, we introduce an optimization strategy for the de-duplication of the solving in our actual tool. That is, before line 13 of Algorithm 1, a repetitive query is performed for the CRC64 [1] [21] value of model state and uncovered metric that will be solved. If it is found that the same solving has been performed before, the solving result can be obtained directly. These solving records are stored as files, which can speed up the test case generation process every time afterward.

Algorithm 1: State Solving

```
Input: Model: The model for test case generation
       TestCases: The test cases from fuzzer
Output: NewTestCase: The test case solved out
1  coverage = ∅  // Record all coverage of the TestCases
2  stateMap = ∅  // The key is test case, the value is state
3  for t in TestCases do
4      cov, state = Model.run(t)  // Get coverage and state
5      coverage = merge(coverage, cov)
6      stateMap[t] = state
7  covMetrics = Model.getAllCovMetric()
8  for m in covMetrics do
9      if m in coverage then
10         continue
11     // Traverse all uncovered metrics
12     for t, state in stateMap do
13         Model.setState(state)  // Reset model state
14         res = solve(Model, m)  // Solving for an uncovered metric
             on a specific model state
15         if res == NULL then
16             // If unsolvable, continue traversing
17             continue
18         NewTestCase = t.addToBack(res)
19         return NewTestCase
```

3.4 Collaborative Execution Strategy

As shown in Figure 6, HybridTCG uses a glue tool to achieve cooperation between a coverage-guided fuzzer and a state-solving engine. First, the glue tool starts fuzzer and then fetches all test cases from the fuzzer corpus. Next, the glue tool executes the test cases in order of length, from short to long, and cumulatively counts the coverage metrics. The test cases that trigger new coverage are provided to the state-solving engine as refined data. Test cases that generate new coverage are also output. When a new test case is generated by the state-solving engine, it is fed back into the fuzzer corpus. Afterward, the process from obtaining test cases from the fuzzer to the solver outputting test cases is repeated. It will not stop until all metrics are covered or a timeout occurs. Note that, the number of test cases generated by the fuzzer at the beginning does not affect the coverage results, because the state-solving engine can be started multiple times.

[1]A fast algorithm for calculating data checksum values. In this paper, it is used to quickly check for duplication of model internal states.

Table 2: A simple abstraction of the coverage metrics for the example model in Figure 1

ID	Metric	Description	State Required	Difficulty for fuzzer	Difficulty for solver
1	Register Success	Connect a port to Switch.	None	Easy	Easy
2	Register Failure	A port can not connect to Switch.	All port slots are occupied.	Easy	Hard
3	Send Success	Send message between two ports.	At least two ports are registered.	Hard	Hard
4	Send Failure (A)	Src port or Dst port not found.	None	Easy	Easy
5	Send Failure (B)	The Dst port is busy.	The Dst port is in busy status.	Hard	Hard
6	Check Success	The checked port is connected.	At least one port is registered.	Easy	Hard
7	Check Failure	The checked port is not connected.	None	Easy	Easy
8	Logout Success	A connected port logout.	At least one port is registered.	Easy	Hard
9	Logout Failure	The target port does not exist.	None	Easy	Easy

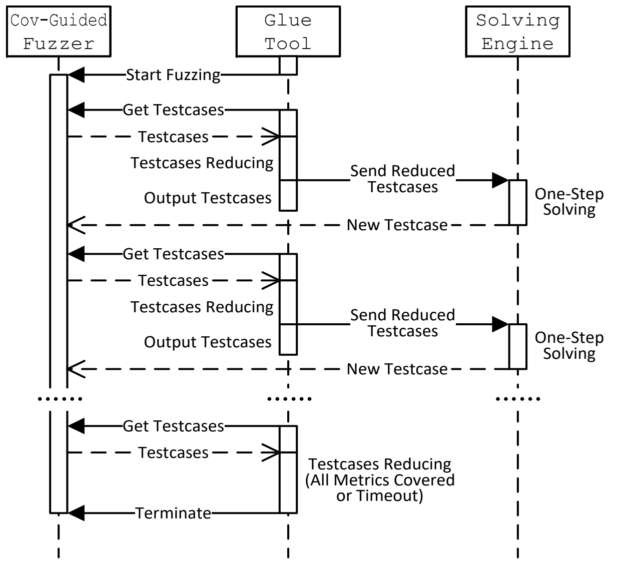


Figure 6: The way HybridTCG works represented by UML timing diagram. Where the glue tool is used to perform the collaboration between the fuzzer and solving engine.

Running Example. To understand more clearly how HybridTCG works, we use the model in Figure 1 here for a running example. But before that, we need to give a further introduction to the model as well as a simple abstraction of the coverage metrics. Figure 1 shows a LANSwitch model, which is mainly used for registering, checking, and logging out of network ports and sending data between two ports. The model provides five inports to perform these operations, which are $\langle PortId, CMD, Arg1, Arg2, Arg3\rangle$. Below is the sample input data for these operations.

- ⟨0x07, 0x2711, 0x00, 0x00, 0x00⟩: Register the port 7.
- ⟨0x07, 0x2713, 0x00, 0x00, 0x00⟩: Check port 7 status.
- ⟨0x07, 0x2714, 0x00, 0x00, 0x00⟩: Logout port 7.
- ⟨0x07, 0x2712, 0x09, 0x10, 0x00⟩: P7 sends data to P9.

We have abstracted some coverage metrics from the LANSwitch model, as shown in Table 2. Note that, these coverage metrics are not concrete evaluation metrics like MCDC, they are just some abstract concepts to represent the model behavior. For instance, metric 1 signifies that a port was successfully registered, metric 3 denotes that data was successfully sent between two ports, and metric 5 implies that data could not be successfully sent due to a busy destination port. The last two columns of Table 2 indicate how easy or hard these metrics are for each of the fuzzer and solver.

For the fuzzer, those metrics that only require changing a single inport data are easy to trigger. This is because the fuzzer's ability to rapidly and systematically mutate values allows the data to quickly meet the conditions. Additionally, since the fuzzer executes continuously, it is also easy for it to further mutate the already mutated data to trigger a value that satisfies another condition. For example, metric 1 is easy because it only needs the CMD port to be 0x2711 and the rest of the ports to be random data. Metric 2 is easy because it only requires the register operation to be repeated a few times. Metric 3 is hard because it not only needs the CMD port to be 0x2712, but it also needs Arg1 to be an already registered port ID. For the solver, solving those metrics that require the model to be in a specific state to be triggered is hard. This is because the complexity of the problem faced by the solver in solving multiple steps is exponential. For example, metric 3 requires at least 3 steps to trigger, registering port 1, registering port 2, and port 1 sending data to port 2. Next, we exemplify how to cover these metrics in Table 2 using our model fuzzing and state-solving approach.

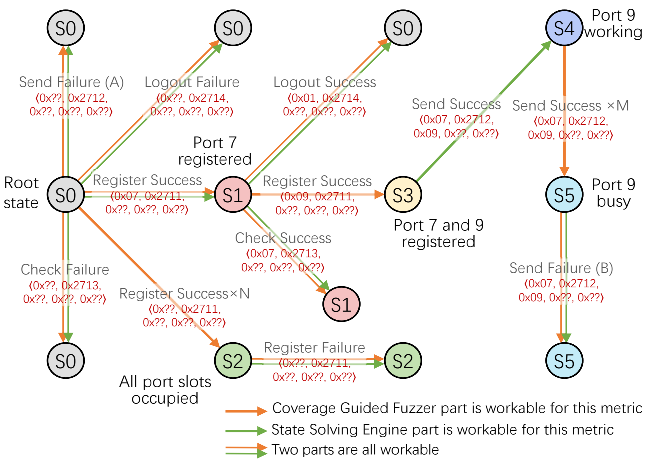

S0
S0
S0
S4
Port 9 working
Send Failure (A)
⟨0x??, 0x2712, 0x??, 0x??, 0x??⟩
Logout Failure
⟨0x??, 0x2714, 0x??, 0x??, 0x??⟩
Logout Success
⟨0x01, 0x2714, 0x??, 0x??, 0x??⟩
Send Success
⟨0x07, 0x2712, 0x09, 0x??, 0x??⟩
Send Success ×M
⟨0x07, 0x2712, 0x09, 0x??, 0x??⟩
Port 7 registered
Root state
S0
Register Success
⟨0x07, 0x2711, 0x??, 0x??, 0x??⟩
S1
Register Success
⟨0x09, 0x2711, 0x??, 0x??, 0x??⟩
S3
Port 7 and 9 registered
S5
Port 9 busy
Check Success
⟨0x07, 0x2713, 0x??, 0x??, 0x??⟩
Check Failure
⟨0x??, 0x2713, 0x??, 0x??, 0x??⟩
Register Success×N
⟨0x??, 0x2711, 0x??, 0x??, 0x??⟩
S1
Send Failure (B)
⟨0x07, 0x2712, 0x09, 0x??, 0x??⟩
S0
S2
Register Failure
⟨0x??, 0x2711, 0x??, 0x??, 0x??⟩
S2
S5
All port slots occupied
Coverage Guided Fuzzer part is workable for this metric
State Solving Engine part is workable for this metric
Two parts are all workable

Figure 7: A test case generation example of collaboration between coverage guided fuzzer part and state solving engine part, corresponding to the LANSwitch model in Figure 1. Where circles with the same color represent the same model state. The text in red color responds the test case row(s).

Figure 7 illustrates the collaborative process between the two parts (for short fuzzer and solver) of HybridTCG. We use a tree diagram to show the model states and the one- or multi-step inputs that can reach these states. The tree nodes indicate model-specific states, with identical states represented by the same color. The edges of the tree indicate model inputs and which coverage metric that input can trigger. Firstly, in the root state, the four metrics Register Success, Send Failure (A), Check Failure, and Logout Failure are available for fuzzer and solver to generate the corresponding test cases. Note that our solver can operate on the root state, which is equivalent to providing it with empty test case data. The Register Failure metric requires all port slots of the model to be occupied to trigger. It is easy for the fuzzer to reach this state, it only needs to repeat the register operation several times. On the contrary, it is hard for the solver, as it faces the problem of state-space explosion when solving for multiple steps as opposed to solving for one step.

Next, suppose we obtain a test case for registering port 7 and obtain the state S1, i.e., port 7 was successfully registered. Based on the state S1 and the existing test case corpus, it is easy for fuzzer to cover Logout Success and Check metrics. It is also easy for the solver to solve for one-step input on the S1 state to achieve them. Due to the fuzzer's randomness, suppose it again randomly generates the operation to register port 9 on top of S1, we obtain the state S3. Note that the state S3 cannot be reached by the solver because Register Success is an already covered metric. To trigger the Send Success metric afterward, we can only do it via solver. Because the metric requires the parameter Arg1 to specify a port ID that has already been registered, which is hard for fuzzer. Solver can directly solve for the input data required to trigger Send Success in state S3 and then obtain state S4 (Port 9 working). To trigger the Send Failure (B) metric, Port 9 needs to be in a busy status. Here again, we need to rely on the fuzzer's fast random generation capability to repeat sending data to port 9 multiple times as a way to reach state S5 (Port 9 busy). Finally, based on S5, another data send is performed to trigger the Send Failure (B) metric. As can be discerned from the path S0->S1->S3->S4->S5 in Figure 7, it is the collaboration between the fuzzer and solver that facilitates the coverage of metrics that would otherwise be arduous for each method.

4 EVALUATION

4.1 Tool Implementation.

HybridTCG[2] is implemented in C++, comprising 37,087 lines of code. In the fuzzing code generation module, we have leveraged the Simulink Embedded Coder for the generation of the model logic. All subsequent processes are automated by code. The coverage instrumentation code we designed is inserted into the main logic code of the model. In the coverage-guided fuzzer part, we implement the model input mutation function based on LibFuzzer [13]. In this way, the three parts of the instrumented logic code, the fuzz driver and the modified LibFuzzer can be compiled together to form a complete coverage-guided fuzzer. For the state-solving engine part, we utilize the CBMC tool to convert the code of the Simulink model into SAT statements, and then employ the MiniSAT constraint solver to implement one-step solving[9]. For setting the model state before solving, we implemented this by initializing the values of the model global variables, since model states are stored using global variables in the model. Next, as shown earlier in Figure 6, we also implemented a glue tool for collaboration between the fuzzer and the solver. Furthermore, we additionally implemented a test case converter, which transposes the binary test cases output by HybridTCG into a Simulink-readable Excel file format. This allows us to use Simulink's coverage statistics feature to make fair comparisons with other works [24].

[2]The implementation and the benchmark models are uploaded on the anonymous website: https://anonymous.4open.science/r/HybridTCG-7514.

4.2 Experiment Setup.

To evaluate the effectiveness of HybridTCG, we conducted comparative experiments with Simulink's built-in verification toolkit, SLDV, and two academic tools, SimCoTest and STCG, on the coverage results. Since other academic and commercial tools are not publicly accessible, we can not compare HybridTCG with them. In addition, the experimental results were analyzed in more depth to illustrate the validity of the methodology of this paper. All experiments are conducted on the same environment (Windows 10, Intel i7-8550U CPU, 16GB RAM, Simulink 2022b, SLDV 4.8) with the same duration (1 hour, the results for 24 hours were the same as for 1 hour). We repeat the experiment 10 times to obtain the average coverage result for a fair comparison. Note that, the final coverage results are almost same of each tool. Although some of the tools contain random strategy, the evaluation results are stable over an hour. All benchmark models are derived from publicly available model sets, which are deployed in embedded scenarios[37]. Table 3 shows the details of these models, including model functionality, number of branches, and number of blocks.

Table 3: The description of benchmark models

Model	Functionality	#Branch	#Block
CPUTask	AutoSAR CPU task dispatch system	107	275
SVPWM	Space vector pulse width modulation	916	1233
TWC	Train wheel speed controller	80	214
NICProtocol	Vehicle NIC communication protocol	46	294
LANSwitch	LAN Switch controller	131	570
UTPC	Underwater thruster power control	92	214
LEDLC	LED matrix load control	94	270
TCP	TCP three-way handshake protocol	146	330
RAC	Robotic arm controller	179	667
TCS1	Control model dedicated to testing	1597	1742
TCS2	Control model dedicated to testing	2752	7196
FMTM	Factory Multi-point Temperature Monitor	89	152

4.3 Evaluation on Coverage Rate.

We use the most widely used Decision Coverage, Condition Coverage and MCDC to measure the effectiveness of different tools in generating test cases[1, 24]. A higher coverage metric signifies a more comprehensive examination of the model, while earlier achievement of coverage indicates more efficient testing.

Table 4 shows the coverage of the test cases generated by the different tools for benchmark models. Compared to SLDV, SimCoTest and STCG, HybridTCG improves the Decision Coverage for 54%, 108% and 24%, Condition Coverage for 50%, 62% and 6%, and MCDC

Table 4: Comparison of the test coverage of different tools

Model	Tool	Decision Coverage	Condition Coverage	MCDC
CPUTask	SLDV	89%	72%	42%
	SimCoTest	72%	56%	21%
	STCG	**100%**	**100%**	**100%**
	HybridTCG	**100%**	**100%**	**100%**
SVPWM	SLDV	45%	55%	-
	SimCoTest	34%	75%	-
	STCG	17%	63%	-
	HybridTCG	**63%**	**96%**	-
TWC	SLDV	46%	68%	40%
	SimCoTest	15%	57%	20%
	STCG	92%	97%	**100%**
	HybridTCG	**96%**	**98%**	**100%**
NICProtocal	SLDV	75%	83%	10%
	SimCoTest	30%	43%	33%
	STCG	95%	98%	**100%**
	HybridTCG	**100%**	**100%**	**100%**
LANSwitch	SLDV	72%	76%	15%
	SimCoTest	78%	81%	15%
	STCG	**100%**	**98%**	**55%**
	HybridTCG	**100%**	**98%**	**55%**
UTPC	SLDV	44%	59%	44%
	SimCoTest	40%	58%	44%
	STCG	**100%**	**100%**	**100%**
	HybridTCG	**100%**	**100%**	**100%**
LEDLC	SLDV	55%	41%	43%
	SimCoTest	55%	41%	43%
	STCG	**98%**	**100%**	**100%**
	HybridTCG	**98%**	**100%**	**100%**
TCP	SLDV	63%	64%	33%
	SimCoTest	82%	74%	17%
	STCG	**99%**	**100%**	67%
	HybridTCG	**99%**	**100%**	**83%**
RAC	SLDV	64%	71%	12%
	SimCoTest	71%	76%	12%
	STCG	98%	98%	23%
	HybridTCG	**100%**	**99%**	**73%**
TCS1	SLDV	60%	65%	16%
	SimCoTest	86%	68%	28%
	STCG	**99%**	85%	41%
	HybridTCG	**99%**	**96%**	**99%**
TCS2	SLDV	79%	83%	43%
	SimCoTest	79%	87%	56%
	STCG	85%	**94%**	92%
	HybridTCG	**86%**	**94%**	**98%**
FMTM	SLDV	76%	77%	25%
	SimCoTest	64%	55%	15%
	STCG	95%	95%	35%
	HybridTCG	**96%**	**96%**	**70%**
Average Improvement	vs SLDV	↑ 54%	↑ 50%	↑ 291%
	vs SimCoTest	↑ 108%	↑ 62%	↑ 282%
	vs STCG	↑ 24%	↑ 6%	↑ 45%

* Since SVPWM does not contain AND and OR logic, it does not involve the MCDC metric.

for 291%, 282% and 45%, respectively. From the table, we can see that HybridTCG achieves higher coverage compared to the other tools. For example, HybridTCG achieves 100% Decision Coverage, 100% Condition Coverage and % MCDC on three models, CPUTask, NICProtocal and UTPC.

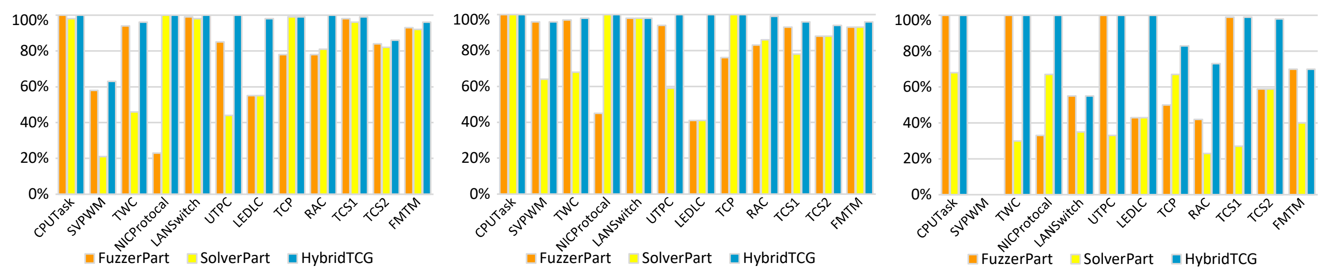
We carefully investigated the models for which we achieved substantially higher coverage that other tools did not, and we found that there was more complex logic in these models than in other models. For example, the SVPWM model contains a large number of numerical judgments and much deeper logic that can only be triggered in very demanding model states. Solving-based methods such as SLDV and STCG are limited by solving complexity and have difficulty exploring the model states on which they rely. For SimCoTest, random search has difficulty triggering model branches that require sophisticated constraints. In contrast, HybridTCG uses fuzzing methods to quickly explore the state space and accurately achieves deep branch coverage using constraint solving. Note that the SVPWM model has a maximum decision coverage of only 63%, which is due to the model's configuration parameters resulting in some branches not being executed.

In addition, we can discern from Table 4 that HybridTCG achieves a much higher MCDC improvement than Decision Coverage and Condition Coverage compared to other tools. This is due to our instrumentation of the MCDC metrics in the coverage-guided fuzzer and the accurate solving of each metric in the state-solving engine.

4.4 Ablation Study on Fuzzer and Solver Part

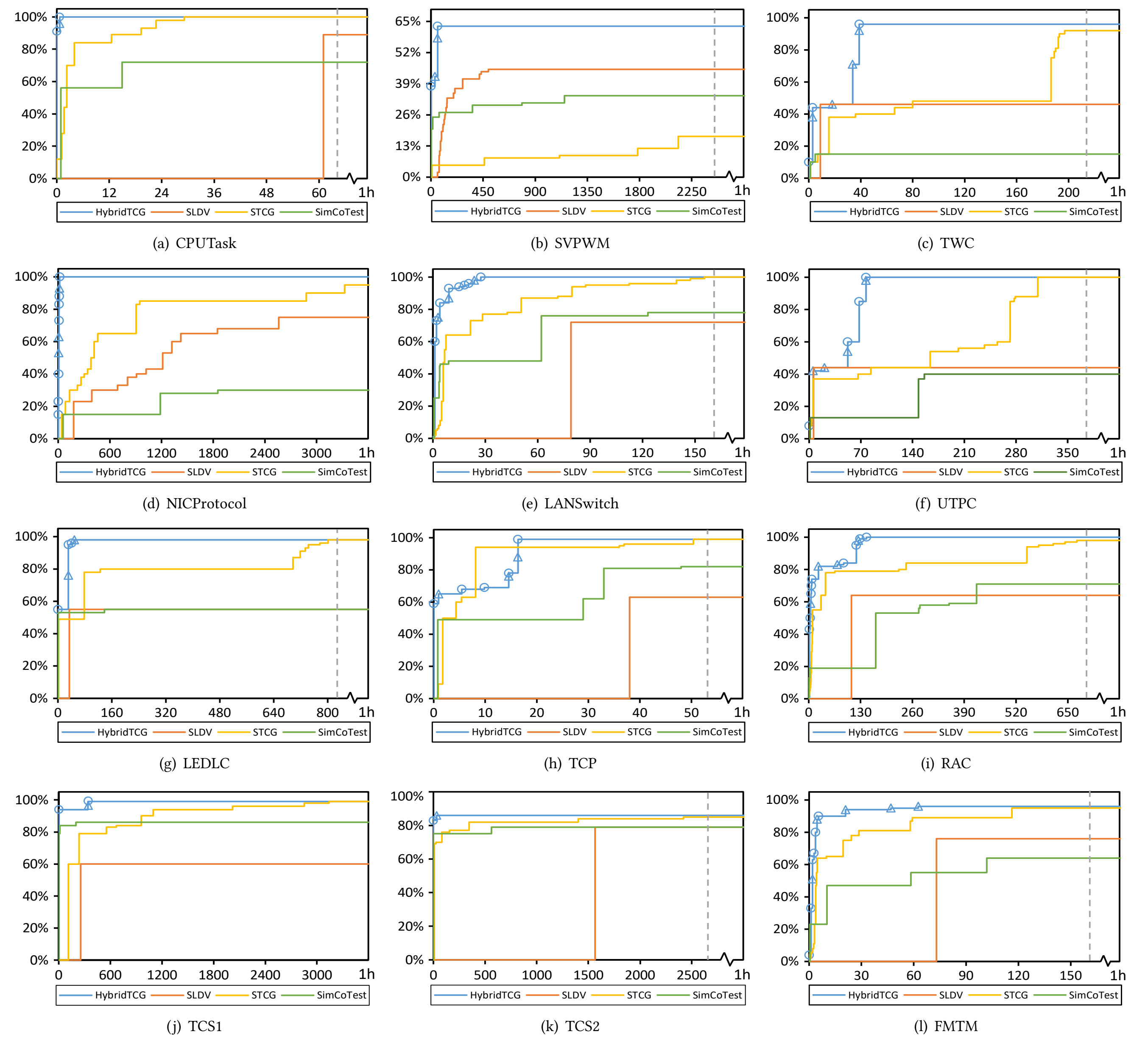
We ran the fuzzer part and solver part separately for the ablation experiment. The specific experimental setup is as follows: For the coverage-guided fuzzer, we let its corpus be generated and used only by itself. At the same time, we store all test cases that generate new coverage and the corresponding timestamps. For the state-solving engine, we let it start solving from the model initialization state. Whenever a new test case is obtained, we save it. Immediately afterward, the saved test cases are fed directly into the solver part for execution. The execution of the state-solving engine is looped through in this way. We also record the timestamp of each test case.

Figure 8 shows the comparison of Decision Coverage, Condition Coverage and MCDC for the fuzzer part, solver part and HybridTCG respectively. Comparing the fuzzer part, HybridTCG is on average 42%, 28% and 49% higher in Decision Coverage, Condition Coverage and MCDC. Despite the same coverage results on the CPUTask model, HybridTCG is 7.1x more efficient. HybridTCG has better coverage results compared to the solver part for all benchmark models,



(a) Decision Coverage (b) Condition Coverage (c) MCDC

Figure 8: Ablation study on fuzzer part (coverage-guided fuzzer) and solver part (state solving engine) of HybridTCG



(a) CPUTask (b) SVPWM (c) TWC
(d) NICProtocol (e) LANSwitch (f) UTPC
(g) LEDLC (h) TCP (i) RAC
(j) TCS1 (k) TCS2 (l) FMTM

Figure 9: The folded line plot of the Decision Coverage versus time. The X-axis is time (s) and the Y-axis is Decision Coverage (%). "○" indicates that this coverage boost is achieved by the coverage-guided fuzzer. "△" indicates that this coverage boost is achieved by the state solving engine.

it is on average 46%, 30% and 125% higher in Decision Coverage, Condition Coverage and MCDC. These results mean HybridTCG can trigger metrics that can not covered by those two parts.

4.5 Efficiency of HybridTCG

The capability of coverage is paramount, as is the efficiency of generating such test cases. We plotted a folded line graph of Decision Coverage versus time by selecting the median coverage efficiency in each model's experiments. In Figure 9, the X-axis is time (s) and the Y-axis is Decision Coverage (%). We can see from Figure 9 that HybridTCG's test case generation efficiency is also higher than other tools, and HybridTCG always generates more coverage than the other three tools.

To analyze the underlying reasons behind HybridTCG's high efficiency and high coverage in depth, we recorded the source of each test case's output, i.e. whether it came from the coverage-guided fuzzer or the state-solving engine. As shown in Figure 9, "○" indicates that this coverage boost is achieved by the coverage-guided fuzzer. "△" indicates that this coverage boost is achieved by the state-solving engine. We can see that the two parts mostly alternate in generating test cases, which suggests that they both play a pretty important role. Looking more closely, we can see that in many subgraphs there is a circular marker "○" directly above a triangular marker "△". This phenomenon indicates that as soon as the state-solving engine generates a new test case, the coverage-guided fuzzer immediately generates a higher coverage. For example, when `HybridTCG` in the CPUTask model triggered about 95% coverage using the state-solving engine, the coverage-guided fuzzer quickly generated test cases that achieved 100% coverage. This observation substantiates the fact that the collaboration between the two parts of `HybridTCG` is efficacious.

An interesting observation in Figure 9 is that for the NICProtocol model, the efficiency of `HybridTCG` in generating test cases is much higher than for the other three tools. This is because this model has a remarkably large number of internal states. This has a huge impact on the performance of the other three tools, particularly SLDV, whereas `HybridTCG`'s state-solving engine automatically removes states that are irrelevant to the constraints during the formal transformation of the model. This allows the state-solving engine to have a faster solving speed.

4.6 Comparison with Software Testing Methods

There is a lot of work on test case generation, especially for software or code. Therefore, advanced testing work in the software domain can be used for testing the models. Three well-known testing tools, Libfuzzer, Driller and KLEE, were selected for additional experiments[7, 13, 34]. Libfuzzer is an in-process coverage-guided fuzzing tool. Driller is a concolic testing tool that combines fuzzing with symbolic execution. KLEE is a symbolic execution engine built on top of the LLVM compiler infrastructure. It is worth noting that our hybrid approach of fuzzing and solving does not use solver to obtain the complete input data as in the case of the constraint execution approach. `HybridTCG` uses the state space obtained from the fuzzer to solve test cases for model steps, which is more concerned with the iterative nature of model execution. We used the model-generated code for testing. We utilized the fuzz driver of `HybridTCG` so that the testing tool could automatically read the test files to execute the code. After the test case generation process, we converted the binary test case files to Excel format to provide coverage statistics for Simulink. The results are shown in Figure 10. Compared to Libfuzzer, `HybridTCG` was on average 173%, 103% and 690% higher in Decision Coverage, Condition Coverage, and MCDC, respectively. Compared to Driller, `HybridTCG` was on average 216%, 153% and 512% higher, respectively. Compared to KLEE, `HybridTCG` was on average 147%, 85% and 431% higher, respectively.

We have analyzed in depth the reason why software testing methods do not work well on models. The main bottleneck is that many of the coverage metrics required for testing on models are discarded by software testing methods. For example, the judgment logic of the truth table for AND and OR blocks. The true and false values of the input data for these boolean blocks are Condition Coverage metrics. They are more concerned with capturing the coverage of blocks of code. The results of the fuzzer part of our ablation study, which included model coverage metrics, also confirm this conclusion.

5 DISCUSSION

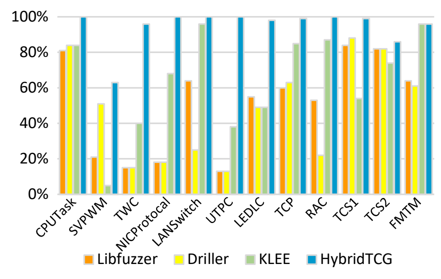
5.1 The Challenge of Permanently False Metrics

In our tools, both the coverage-guided fuzzer and the state-solving engine can only check the satisfiability of metrics. However, it is not possible to identify which metrics are always uncoverable, i.e., permanently false metrics. This leads to the fact that the state-solving engine may solve for a permanently false metric on many states, which consequently wastes a lot of time. This is even more so for the fuzzer. And once the remaining uncovered metrics are only those that are permanently false, we have no way of knowing when it is time to terminate `HybridTCG`. This also leads to a meaningless execution of the fuzzer and solver after that.

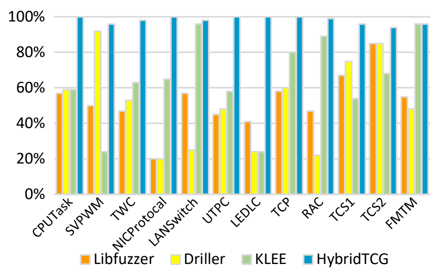
We currently do not have a fully effective solution to deal with this complex issue. However, we have devised simple strategies to mitigate this problem: for those metrics that cannot be solved multiple times, we try stateless solving, where the model state is set to indeterminate values to be solved. If the metric still cannot be solved, it is considered to be permanently false. This is equivalent to the metric being unsatisfiable in any state. Whereas if the metric can be solved, we cannot conclude that it is satisfiable. This is because we cannot determine whether the model state on which the metric depends can be reached by executing multiple input steps.

5.2 Effect of Initial Parameters of the Model

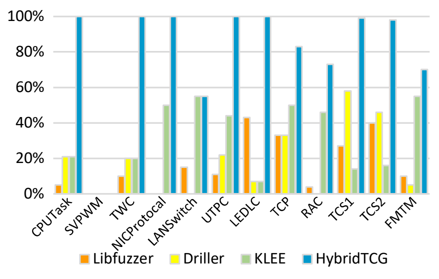
In our experiments, we found that some models would be set with initial parameters before execution, which resulted in some of the logic in the model being restricted by parameter conditions and not being able to be executed. This means that different model initial parameters can enable different model coverages to be triggered. The two benchmark models, SVPWM and TCS2, are strongly influenced by the initial parameters.



(a) Decision Coverage



(b) Condition Coverage



(c) MCDC

Figure 10: Comparison of coverage with test case generation methods in the software domain. Libfuzzer is an in-process coverage-guided fuzzing tool. Driller is a concolic testing tool that combines fuzzing with symbolic execution.

However, as we know from our industry partners, the initial parameters of the model are often used as part of the test cases. For this reason, we have added a simple method to generate the initial parameters in the HybridTCG. It reads the corresponding data from the binary stream at the beginning of the fuzz drive as initial parameters. The subsequent data are then the inputs for each step of the model. Each time the state solver engine dynamically executes a test case, it first reads and sets the initial parameters for the model. In this approach, we re-ran the test case generation with initial parameters for the SVPWM and TCS2 models. As a result, they obtained more than 95% coverage on all metrics.

5.3 One-Setp Solving vs Multi-Step Solving

We experimented on the performance of multi-step solving. For the benchmark models, two-step solving and three-step solving take 3x and 6x longer, respectively, than one-step solving. We tried two strategies for applying multi-step solving, including "try multi-step solving immediately when one-step solving fails" and "try multi-step solving when one-step solving fails on all model states". The experimental results show that both strategies cause significant performance overhead (the efficiency was reduced by 180% and 40%, respectively), but a limited improvement in coverage. Only the TWC model improved its decision coverage from 96% to 98%.

From the results, it appears that there are some coverage metrics that need to be solved for multiple steps to be triggered. However, the introduction of multi-step solving does not bring significant benefits to HybridTCG, but rather has significant side effects. Based on the methodological analysis, we believe that the fuzzing part of HybridTCG has reached the coverage exploration capability of multi-step solving. This is because once the solving engine has explored a new coverage, the fuzzer will mutate the new test case to explore more coverage. This can be confirmed by the folded line plot in Figure 9.

6 CONCLUSION

In this paper, HybridTCG is proposed to optimize the test case generation for Simulink models, especially for the control models that have complex logic and internal states. More specifically, the coverage-guided fuzzer uses model-specific fuzz driver generation, coverage metrics instrumentation, and model input mutation to generate test cases quickly and consistently. The state-solving engine solves one step input further based on the state explored in the test cases output by the fuzzer. These two parts collaborate to achieve efficient and high-coverage test case generation. Experiments show that HybridTCG can perform well on benchmark Simulink models. Compared to SLDV, SimCoTest and STCG, HybridTCG achieves improvements in Decision Coverage, Condition Coverage and MCDC. Not only that, it is also much more efficient than other tools. We demonstrate the effectiveness of the collaboration between the fuzzer part and the solver part through ablation experiments. We also compare HybridTCG with the testing work in the software domain, and the experimental results show that our method is more suitable for model testing. Our future work will address the permanently false metrics and the metrics with specific pre-conditions to facilitate the efficiency of HybridTCG.

7 ACKNOWLEDGEMENTS

This research is sponsored in part by the National Key Research and Development Project (No. 2022YFB3104000), China Postdocoral Science Foundation (BX20230183, 2023M731954) and NSFC Program (No. 92167101, 62021002, 62372263).

REFERENCES

[1] William Aldrich. 2002. Using model coverage analysis to improve the controls development process. In *AIAA Modeling and Simulation Technologies Conference and Exhibit*. 4684–4694.

[2] Saswat Anand, Edmund K Burke, Tsong Yueh Chen, John Clark, Myra B Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn, Antonia Bertolino, et al. 2013. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software* 86, 8 (2013), 1978–2001.

[3] Aitor Arrieta, Shuai Wang, Urtzi Markiegi, Goiuria Sagardui, and Leire Etxeberria. 2017. Search-based test case generation for cyber-physical systems. In *2017 IEEE Congress on Evolutionary Computation*. 688–697.

[4] Tansu Zafer Asici, Burak Karaduman, Raheleh Eslampanah, Moharram Challenger, Joachim Denil, and Hans Vangheluwe. 2019. Applying model driven engineering techniques to the development of contiki-based IoT systems. In *Proceedings of the 1st International Workshop on Software Engineering Research & Practices for the Internet of Things*. IEEE Press, 25–32.

[5] Ezio Bartocci, Niveditha Manjunath, Leonardo Mariani, Cristinel Mateis, and Dejan Ničković. 2021. CPSDebug: Automatic failure explanation in CPS models. *International Journal on Software Tools for Technology Transfer* (2021), 1–14.

[6] Axel Belinfante, Lars Frantzen, and Christian Schallhart. 2005. 14 tools for test case generation. In *Model-based testing of reactive systems*. Springer, 391–438.

[7] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, December 8-10, San Diego, California, USA*. USENIX Association, 209–224.

[8] Yaohui Chen, Mansour Ahmadi, Reza Mirzazade Farkhani, Boyu Wang, and Long Lu. 2020. MEUZZ: Smart Seed Scheduling for Hybrid Fuzzing. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses, San Sebastian, Spain, October 14-15*. USENIX Association, 77–92.

[9] Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems (Lecture Notes in Computer Science, Vol. 2988)*, Kurt Jensen and Andreas Podelski (Eds.). Springer, 168–176.

[10] Rance Cleaveland, Scott A Smolka, and Steven T Sims. 2006. An Anstrumentation-Based Approach to Controller Model Validation. In *Automotive Software Workshop*. Springer, 84–97.

[11] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. 2017. SCADE 6: A formal language for embedded critical software development. In *11th International Symposium on Theoretical Aspects of Software Engineering*. 1–11.

[12] Frank Elberzhager, Alla Rosbach, and Thomas Bauer. 2013. Analysis and testing of matlab simulink models: a systematic mapping study. In *Proceedings of the 2013 International Workshop on Joining AcadeMiA and Industry Contributions to Testing Automation*. 29–34.

[13] Google. 2016. LibFuzzer Tutorial. https://github.com/google/fuzzing/blob/master/tutorial/libFuzzerTutorial.md. Accessed: 2024-03-29.

[14] Reinhard Hametner, Benjamin Kormann, Birgit Vogel-Heuser, Dietmar Winkler, and Alois Zoitl. 2011. Test case generation approach for industrial automation systems. In *The 5th international conference on automation, robotics and applications*. IEEE, 57–62.

[15] Grégoire Hamon, Bruno Dutertre, Levent Erkok, John Matthews, Daniel Sheridan, David Cok, John Rushby, Peter Bokor, Sandeep Shukla, Andras Pataricza, et al. 2008. Simulink Design Verifier-Applying Automated Formal Methods to Simulink and Stateflow. In *Third Workshop on Automated Formal Methods*.

[16] Nannan He, Philipp Rümmer, and Daniel Kroening. 2011. Test-Case Generation for Embedded Simulink via Formal Concept Analysis. In *Proceedings of the 48th Design Automation Conference*. 224–229.

[17] Itti Hooda and Rajender Chhillar. 2014. A review: study of test case generation techniques. *International Journal of Computer Applications* 107, 16 (2014), 33–37.

[18] SRI International. [n. d.]. *Symbolic Analysis Laboratory*. https://sal.csl.sri.com/

[19] Karim Jahed and Juergen Dingel. 2019. Enabling model-driven software development tools for the internet of things. In *Proceedings of the 11th International Workshop on Modelling in Software Engineerings*. IEEE Press, 93–99.

[20] Yu Jiang, Houbing Song, Yixiao Yang, Han Liu, Ming Gu, Yong Guan, Jiaguang Sun, and Lui Sha. 2018. Dependable model-driven development of cps: From stateflow simulation to verified implementation. *ACM Transactions on Cyber-Physical Systems* 3, 1 (2018), 12.

[21] Philip Koopman and Tridib Chakravarty. 2004. Cyclic redundancy code (CRC) polynomial selection for embedded networks. In *International Conference on*

Dependable Systems and Networks, 2004. IEEE, 145–154.

[22] J Krizan, L Ertl, M Bradac, M Jasansky, and A Andreev. 2014. Automatic code generation from MATLAB/Simulink for critical applications. In *2014 IEEE 27th Canadian Conference on Electrical and Computer Engineering (CCECE)*. IEEE, 1–6.

[23] Wenbin Li, Franck Le Gall, and Naum Spaseski. 2018. A survey on model-based testing tools for test case generation. In *Tools and Methods of Program Analysis: 4th International Conference, TMPA 2017, Moscow, Russia, March 3-4, 2017, Revised Selected Papers 4*. Springer, 77–89.

[24] MathWorks. 2023. Simulink Coverage. https://www.mathworks.com/products/simulink-coverage.html. Accessed: 2024-03-29.

[25] Reza Matinnejad, Shiva Nejati, Lionel C Briand, and Thomas Bruckmann. 2016. SimCoTest: A test suite generation tool for Simulink/Stateflow controllers. In *Proceedings of the 38th International Conference on Software Engineering Companion*. 585–588.

[26] Reza Matinnejad, Shiva Nejati, Lionel C Briand, and Thomas Bruckmann. 2018. Test generation and test prioritization for simulink models with dynamic behavior. *IEEE Transactions on Software Engineering* 45, 9 (2018), 919–944.

[27] Swarup Mohalik, Ambar A Gadkari, Anand Yeolekar, KC Shashidhar, and S Ramesh. 2014. Automatic test case generation from Simulink/Stateflow models using model checking. *Software Testing, Verification and Reliability* 24, 2 (2014), 155–180.

[28] Cu D Nguyen, Alessandro Marchetto, and Paolo Tonella. 2012. Combining model-based and combinatorial testing for effective test case generation. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. 100–110.

[29] Jan Peleska, Elena Vorobev, and Florian Lapschies. 2011. Automated test case generation with SMT-solving and abstract interpretation. In *NASA Formal Methods: Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings 3*. Springer, 298–312.

[30] Florian Rademacher, Jonas Sorgalla, Sabine Sachweh, and Albert Zündorf. 2019. A model-driven workflow for distributed microservice development. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. ACM, 1260–1262.

[31] Manoranjan Satpathy, Anand Yeolekar, and S Ramesh. 2008. Randomized directed testing (REDIRECT) for Simulink/Stateflow models. In *Proceedings of the 8th ACM international conference on Embedded software*. 217–226.

[32] Simulink and Matlab. [n. d.]. *Simulink Documentation*. https://www.mathworks.com/help/simulink/index.html

[33] Sergey Staroletov, Nikolay Shilov, Vladimir Zyubin, Tatiana Liakh, Andrei Rozov, Ivan Konyukhov, Innokenty Shilov, Thomas Baar, and Horst Schulte. 2019. Model-driven methods to design of reliable multiagent cyber-physical systems. In *Proc. of the Conference on Modeling and Analysis of Complex Systems and Processes*.

[34] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Annual Network and Distributed System Security Symposium*.

[35] Zhuo Su, Dongyan Wang, Yixiao Yang, Yu Jiang, Wanli Chang, Liming Fang, Wen Li, and Jiaguang Sun. 2021. Code Synthesis for Dataflow Based Embedded Software Design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2021).

[36] Zhuo Su, Dongyan Wang, Zehong Yu, Yixiao Yang, Yu Jiang, Rui Wang, Wanli Chang, Wen Li, Aiguo Cui, and Jiaguang Sun. 2022. PHCG: Optimizing Simulink Code Generation for Embedded System with SIMD Instructions. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2022).

[37] Zhuo Su, Zehong Yu, Dongyan Wang, Yixiao Yang, Rui Wang, Wanli Chang, Aiguo Cui, and Yu Jiang. 2023. STCG: State-Aware Test Case Generation for Simulink Models. In *60th ACM/IEEE Design Automation Conference (DAC)*. ACM.

[38] Mingzhe Wang, Jie Liang, Yuanliang Chen, Yu Jiang, Xun Jiao, Han Liu, Xibin Zhao, and Jiaguang Sun. 2018. SAFL: increasing and accelerating testing coverage with symbolic execution and guided fuzzing. In *Proceedings of the 40th International Conference on Software Engineering: Companion ProceeedingsGothenburg, Sweden, May 27 - June 03*. ACM, 61–64.

[39] Zehong Yu, Zhuo Su, Yixiao Yang, Jie Liang, Yu Jiang, Aiguo Cui, Wanli Chang, and Rui Wang. 2022. Mercury: Instruction Pipeline Aware Code Generation for Simulink Models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 11 (2022), 4504–4515.